A Performance Prediction Framework for Data Intensive Applications on Large Scale Parallel Machines[*]

Mustafa Uysal, Tahsin M. Kurc, Alan Sussman and Joel Saltz
UMIACS and Department of Computer Science
University of Maryland
College Park, MD 20742
{uysal,kurc,als,saltz}@cs.umd.edu

Abstract

This paper presents a simulation-based performance prediction framework for large scale data-intensive applications on large scale machines. Our framework consists of two components: application emulators and a suite of simulators. Application emulators provide a parameterized model of data access and computation patterns of the applications and enable changing of critical application components (input data partitioning, data declustering, processing structure, etc.) easily and flexibly. Our suite of simulators model the I/O and communication subsystems with good accuracy and execute quickly on a high-performance workstation to allow performance prediction of large scale parallel machine configurations. The key to efficient simulation of very large scale configurations is a technique called *loosely-coupled simulation* where the processing structure of the application is embedded in the simulator, while preserving data dependencies and data distributions. We evaluate our performance prediction tool using a set of three data-intensive applications.

1 Introduction

In recent years, data-intensive parallel applications [1, 2, 5, 6, 8] have emerged as one of the leading consumers of cycles on parallel machines. The main distinction of these applications from more traditional compute-intensive applications is that they access and perform operations on huge amounts of disk-resident data. It is critical that future parallel machines be designed to accommodate the characteristics of data-intensive applications. Conversely, application developers need tools to predict the performance of their applications on existing and future parallel machines.

In this paper we present a simulation-based framework for performance prediction of large scale data-intensive applications on large scale parallel machines. Our framework consists of two components: *application emulators* and *a suite of simulators*. We have developed application emulators that accurately capture the behavior of three data-intensive applications that represent three typical classes of data-intensive applications, in a sufficient level of detail for performing performance

[*]This research was supported by the Department of Defense, Advanced Research Projects Agency and Office of Naval Research, under contract No. N66001-97-C-8534, by NSF under contracts #BIR9318183, #ACI-9619020 (UC Subcontract #10152408) and #CDA9401151, by ARPA under contract No. #DABT63-94-C-0049 (Caltech subcontract #9503), and by grants from IBM and Digital Equipment Corporation.

prediction for large scale parallel machines. Emulators provide parameterized models of these applications, which make it possible to scale applications in a controlled way. We have also developed a set of simulation models that are both sufficiently accurate and execute quickly, so are capable of simulating parallel machine configurations of up to thousands of processors on a high-performance workstation. These simulators model the I/O and communication subsystems of the parallel machine at a sufficiently detailed level for accuracy in predicting application performance, while providing relatively coarse grain models of the execution of instructions within each processor. We describe a new technique, *loosely coupled simulation*, that embeds the processing structure of the application in the form of *work flow graphs* into the simulator while preserving the application workload. This technique allows accurate, yet relatively inexpensive performance prediction for very large scale parallel machines.

The rest of the paper is organized as follows. Section 2 presents the class of applications for which we have developed application emulators. Section 3 describes in detail what an application emulator is, and presents an application emulator for one of the applications as an example. In Section 4, we present the suite of simulation models we have developed and discuss the various tradeoffs that can be made between accuracy of prediction and speed of simulation. In particular, we discuss the issues involved in coupling application emulators to simulators, and describe loosely-coupled simulation as an efficient technique for interaction between an application emulator and a simulator. Section 5 presents an experimental evaluation of the simulation models. Related work is briefly summarized in Section 6, and we conclude in Section 7.

2 Data Intensive Applications Suite

In this section we briefly describe the data intensive applications that have motivated this work.

2.1 Remote sensing - Titan and Pathfinder

Titan [6] is a parallel shared-nothing database server designed to handle satellite data. The input data for Titan are sensor readings from the entire surface of the earth taken from the AVHRR sensor on the NOAA-7 series of satellites. The satellite orbits the earth in a polar orbit, and the sensor sweeps the surface of the earth gathering readings in different bands of the electro-magnetic spectrum. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded for indexing purposes. In a typical operation for Titan, user issues a query to specify the data of interest in space and time. Data intersecting the query are retrieved from disks and processed to generate the output. The output is a two-dimensional multi-band image generated by various types of aggregation operations on the sensor readings, with the resolution of its pixels selected by the query.

Titan operates on data-blocks, which are formed by groups of spatially close sensor readings. When a query is received, a list of data-block requests for each processor is generated. Each list contains read requests for the data-blocks that are stored on the local disks of the processor and that intersect the query window. The operation of Titan on a parallel machine employs a peer-to-peer architecture. Input data-blocks are distributed across the local disks of all processors and each processor is involved in retrieval and processing of data-blocks. The 2D output image is also partitioned among all processors, and each processor is responsible for processing data-blocks that fall into its local subregion of the image. Processors perform retrieval, processing and exchange of data-blocks in a completely asynchronous manner [6]. In this processing loop, a processor issues disk reads, sends and receives data-blocks to and from other processors, and performs the computation required to process the retrieved data-blocks. Non-blocking I/O and communication operations are used to hide latency and overlap these operations with computation. The data-blocks are the atomic units of I/O and communication. That is, even if a data-block partially intersects with the query window and/or the subregion of the output image assigned to a processor, the entire data-block is retrieved from disk and is exchanged between processors.

Pathfinder [1] is very similar to Titan except that it always processes *all* the input data that is available for a particular time period, over the entire surface of the earth. In addition, the operation of Pathfinder on a parallel machine employs a client/server architecture with separate I/O nodes and compute nodes.

2.2 Virtual Microscope

The Virtual Microscope [8] is designed to emulate the behavior of a high-power light microscope. The input data for the Virtual Microscope are digitized images of full microscope slides under high power. Each slide consists of several focal planes. The output of a query into the Virtual Microscope is a multi-band 2D image of a region of a slide in a particular focal plane at the desired magnification level (less than or equal to the magnification of the input images). The server part of the software running on the parallel machine employs a peer-to-peer architecture. As in Titan and Pathfinder, input data is partitioned into data-blocks and distributed across the disks on the parallel machine. In a typical operation, multiple clients can simultaneously send queries to the server. When a query is received, each processor in the server retrieves the blocks that intersect with the query from its disks, processes these blocks, and sends them to the client. There is no communication or coordination between server processors. Different processors can even operate on different queries at the same time.

3 Application Emulators

An application emulator is a program that, when run, exhibits computational and data access patterns that closely resemble the patterns observed when executing a particular class of applications. In practice, an emulator is a simplified version of the real application, but contains all the necessary communication, computation and I/O characteristics of the application required for the performance prediction study. Using an emulator results in less accurate performance estimations than using full application, but it is more robust and enables fast performance predictions for rapid prototyping of new machines. An application emulator models the computation and data access patterns of the full application in a parameterized way. Adjusting the values of the parameters makes it possible to generate various application scenarios within a single class of applications.

In a simulation-based performance prediction framework, application emulator provides a specification of the behavior of the application to the simulator. Using an application emulator has several advantages over using traces from actual program runs or running the full application on the simulator. First, a trace is static and represents the behavior of the application for a single run on a particular configuration of the machine. Since an application emulator is a program that can be run on the simulator, it can model the dynamic nature of an application and can be used for different machine configurations. Second, running a real application may complicate the task of the simulator unnecessarily. By abstracting away parts of the application that are not critical to predicting performance, an application emulator can allow an efficient simulation without getting bogged down in the unimportant details of the application. Third, execution of a complete application requires the availability of real input data. Since the application behavior is only emulated, an application emulator does not necessarily require real input data, but can also emulate the characteristics of the actual data. This can enable performance studies of applications on large machine configurations with large datasets. Fourth, the level of abstraction in the emulator can be controlled by the user. An application emulator without a great amount of detail can be used for rapid prototyping of the performance of the application on a new machine configuration; while a highly detailed emulator can be used, for instance, to study different parallelization strategies for the application.

In this work we target a framework that enables the studying of large scale applications and large scale machines (consisting of several thousands of processors). For this reason, application emulators developed in this work do not provide very detailed models of applications in order to conduct performance prediction studies in reasonable amount of time. However, they model the salient characteristics of each application class in a parameterized and flexible way, thus making it possible to generate various application scenarios within the same class of applications and to emulate the behavior of application with larger datasets for large scale machines. We now describe the emulator for Titan in more detail.

3.1 Case Study: An Application Emulator for Titan

Titan has three major components that characterize the I/O, communication and processing patterns in this class of applications: *input data set(s)*, *output data set(s)*, and *the processing loop.*

Input data: As we discussed in Section 2.1, Titan operates on data-blocks. Although each data-block contains the same number of input elements (sensor readings), the spatial extent of each data-block varies. This is because of the characteristics of the satellite orbit and the AVHRR sensor, which causes the extent of data-blocks containing sensor readings near the earth's poles to be larger than that of data-blocks near the equator. In addition, there are more spatially overlapping data-blocks near the poles than near equator. Thus, each Titan data-block and the distribution of the data-blocks through the input attribute space are somewhat irregular. The irregular nature of the input data also determines the irregular communication and computation patterns of Titan. In the emulator, a data-block is represented by four parameters. A *bounding rectangle* represents the spatial extent of a data-block. The *disk id*, *offset into the disk*, and *block size* are used to emulate I/O patterns. Synthetic data-blocks are generated using simple parameterized functions. In this way, the number of data-blocks can be scaled for large scale machines quickly, while preserving the important characteristics of such satellite input data, as described above. Using simple functions also allows us to change the input characteristics in a controlled way. A user can control the partitioning of input data into blocks, the number of data-blocks, and the distribution of the data-blocks through the input attribute space to generate different application scenarios.

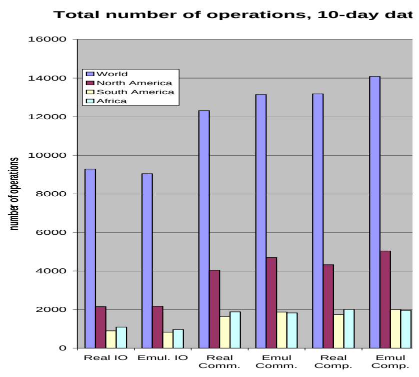
Titan uses Moon's *minimax* algorithm [12] to distribute data-blocks across the disks in the machine. This algorithm achieves good load balance in disk accesses for a wide class of queries. However, it takes a long time to decluster even a moderate number of blocks across a small number of disks [6]. In the emulator, a simple round-robin assignment strategy has been employed to decluster the blocks across the available disks quickly. We are therefore trading off accuracy in modeling the application for efficiency in performing the emulation. Nevertheless, not much accuracy is lost, since we have observed that a round-robin assignment achieves good load balance (but not always as good as the minimax algorithm, especially for queries with small spatial extent).

Output data: Titan implements a workload partitioning strategy. The output, which is a 2D image bounded by the spatial extent of the query in longitude and latitude, is partitioned among processors. Processors are (logically) organized into a 2D grid. This grid is superimposed on the output grid to create rectangular subregions of the output with equal areas. In the emulator, the output is represented by a 2D rectangle bounded by the spatial extent of the input query. The number of processors in each dimension of the 2D processor grid is controlled by the user.

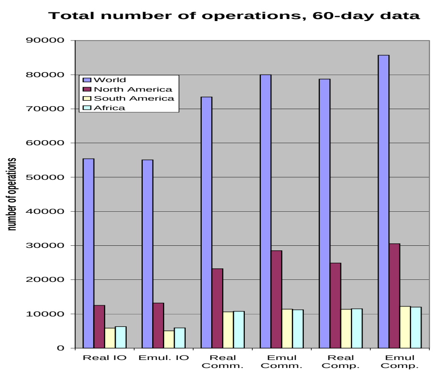
Processing loop: The most important characteristic of the processing loop of Titan is that all disk reads, message sends and receives are non-blocking operations. Despite this seemingly very dynamic structure, there do exist dependencies between operations. First, all message send operations on a data-block depend on the completion of the disk read for that data-block. A completed disk read operation can initiate message send operations to other nodes if the retrieved data-block intersects with their output regions. Moreover, if a data-block intersects with the local output image, then it is enqueued for processing locally. Initiation of receive operations in a processor does not depend directly on the reads or sends in that processor. However, completion of a receive operation depends on the corresponding send operation on the processor that has the data-block on its local disks. When a data-block is received, it is enqueued for processing. The emulator retains the dependencies between operations when generating events for the simulator.

The processing time of a data-block depends on the data values in the data-block. Our simulators model the I/O and communication subsystems of the parallel machine at a sufficiently detailed level, while providing relatively coarse grain models of the execution of instructions within each processor. Therefore, the emulators only have to capture the I/O and communication patterns of the applications accurately, as well as completely capturing the dependency patterns between operations, but do not have to model all the details of the computations performed. Each data-block in our applications is assumed to take the same amount of time to be processed, without regard to the data values it contains. This value is provided to the emulator by the user.

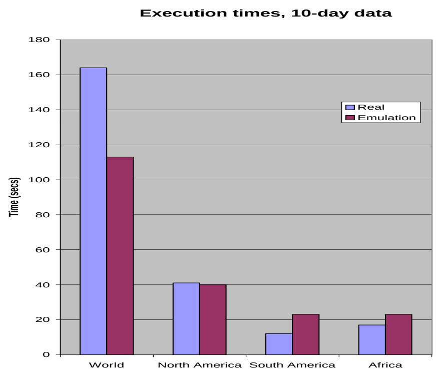
Figures 1(a)-(d) compare the behavior of the Titan emulator with that of the full Titan application. Titan currently runs on 15 nodes of the IBM SP2 at the University of Maryland [6]. Its input data consists of 60 days (24 GBytes) of AVHRR data distributed across the 60 disks attached to the nodes (4 local disks per node). Experiments were carried out using queries covering four regions of the earth: *World*, *North America*, *South America*, *Africa* [6], accessing either 10 or 60 days of data. We should note here that since we target performance prediction for large scale machines, we do not require that the application emulator precisely capture the data access and computation patterns exhibited by the application, but want to capture the behavior of the application when input parameters are changed and when the input dataset size is scaled. Figures 1 (a) and (b) show the total number of I/O, communication and computation operations performed by the Titan emulator and by Titan for the different queries. Figures 1 (c) and (d) show the execution times for the emulator and Titan on the SP2. When the application emulator is executed on the real machine, it performs actual non-blocking read, send and receive operations. However, the processing for a data-block is emulated by a constant delay, which was modeled as the average processing time



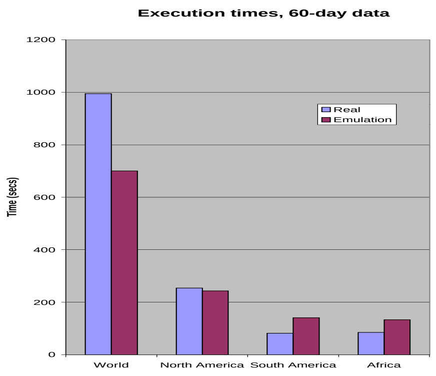
(a)



(b)



(c)



(d)

Figure 1: Comparison of Titan emulator with real Titan in terms of total number of operations for (a) 10-day and (b) 60-day data, and execution time for (c) 10-day and (d) 60-day data.

for data-blocks accessed in several queries by Titan. For this reason, the execution times of the application emulator and the real application may not be close to each other for some queries, such as for the *World* query. However, as is seen from the figures, the application emulator exhibits similar behavior to that of the real application, and consistently matches the application across different queries as well as when the input dataset is scaled.

4 Simulation Models

Our main objective is to develop a simulation framework that is capable of simulating a parallel machine consisting of thousands of processors. We specifically target a data intensive application workload that can process multiple terabytes of data. We require efficient simulators that can run on moderately equipped, readily available hardware, such as a high-performance workstation. The relatively complicated processing structure of the target applications, which seek to overlap computation, communication, and I/O to attain high levels of performance, exacerbates the efficiency problem. A typical simulation of a large scale parallel machine needs to deal with hundreds of thousands of pending I/O operations as well as millions of outstanding messages between the processors. This section addresses these simulation efficiency issues.

4.1 Hardware Models

In order to achieve high levels of efficiency, we employ coarse grain hardware models for the network, disk, I/O bus, and processor. We assume that each node is connected to its peers by dedicated point-to-point links. The time to transfer a message of size L over a network link is modeled as $T = \alpha + L/\beta$ where α and β represent the wire latency and bandwidth, respectively. We neither model the network topology nor link congestion, but we do model end-point congestion that might occur when multiple nodes attempt to transfer messages to a single node. Our disk model consists of four parts: 1) a fixed parameter for the disk controller overhead, 2) a disk bandwidth parameter that specifies the transfer speed to and from the disk media, 3) a disk seek-time parameter that is modeled as a linear function of seek distance, and 4) the rotational position of the disk, which is modeled by calculating how many times the disk would have revolved since the last request was served, at the disk's nominally rated speed. The I/O bus is modeled in the same way as the network links, consisting of a latency and a bandwidth component. When multiple devices contend for the bus, the effects of congestion are modeled. The processor is modeled at a coarse grain; the time interval that the processor was busy computing is the only parameter and it is specified by the application emulator for each operation performed.

Unfortunately, having only coarse grain hardware models is not sufficient to model the performance of data intensive applications for very large configurations. Interaction between application emulators and the hardware simulator also plays an important role in the efficiency of the simulation. In the next two sections we present two different approaches, referred to here as *tightly-coupled simulation* and *loosely-coupled simulation*, for interaction between application emulators and hardware simulators. Both approaches are event-driven. They differ in the granularity of interaction between the simulator and the emulator and the way the interaction occurs.

4.2 Tightly-coupled Simulation

In tightly-coupled simulation, the granularity of interaction between the simulator and the emulator is a single event (e.g., disk read, data-block send). Just as a full application program does, the emulator issues requests one by one to the simulator, emulating the actual behavior of the application with calls to the filesystem for I/O and to the message passing library for interprocessor communication. Our simulator library provides an API for both blocking and non-blocking I/O and communication operations, as well as calls to check the completion of these operations. If the calls are asynchronous calls, such as a non-blocking send, the simulator returns a request id, and the emulator uses that id to check the status of the request later. Each application (emulator) process running on a node of the simulated target machine is implemented by a thread, which is referred to as *emulator thread*. Emulator threads are responsible for simulating the behavior of the applications with respect to the input data. In addition to emulator threads, the combined system has a simulator thread that runs the main simulation engine. It is responsible for processing requests received from emulator threads, keeping track of simulated time, and scheduling emulator threads to make sure that events happen in logical clock order.

4.3 Loosely-coupled Simulation

Interaction between the emulator and the simulator in tightly-coupled simulation closely resembles the interaction between an application and a real machine. However, this approach has several drawbacks which make it unsuitable for simulation of large scale machines. First, the number of emulator threads increases with the number of processors. Handling a large number of threads becomes very costly for the simulator, as it has to schedule emulator threads to ensure correct logical ordering of events. Second, as hardware configurations get larger, message and I/O tables for outstanding non-blocking operations grow very large, becoming very costly to manage and slowing down the network model, which must check for end-point congestion. Moreover, each emulator thread has to maintain local data structures to keep track of outstanding non-blocking operations, replicating the simulator's work and increasing memory requirements. As in a real machine, the status of non-blocking operations is determined by explicitly checking with the simulator; for multiple terabyte datasets, the overheads for these checks become enormous and contribute both to processing time and to the number of context switches between emulator and simulator threads.

To address these efficiency issues, we introduce a technique called *loosely-coupled simulation*. Loosely coupled simulation is currently applicable to applications with a processing loop similar to the one described in Section 2.1, although we are working on applying it to other types of applications. The key idea in loosely-coupled simulation is to embed the processing loop of the application and its dependency structure, modeled by *work flow graphs*, into the simulator. Therefore only two threads are required; a simulator thread and an emulator thread. As in tightly-coupled simulation, the emulator thread is responsible for generating events and the simulator thread is responsible for processing these events. However, unlike tightly-coupled simulation, the emulator and the simulator interact in distinct phases, called *epochs*, rather than interacting at every event.

4.3.1 Modeling the Application Processing Structure: Work Flow Graphs

A *work flow graph* describes the dependencies between operations performed on a single data-block. A node in the graph represents an operation performed on the data-block. In our current set of applications, there are four types of operations: *read*, *send*, *receive*, and *compute*. The directed graph edges represent the dependencies between these operations. For example, an edge from a read operation to a send operation indicates that a send operation on a data-block should start after the read operation is completed. In the applications considered in this work, there are no dependencies between operations performed on different data-blocks neither within the same processor nor across different processors. As a result, work flow graphs impose a partial order on events, and in a sense describe the *life cycle* of a single data-block. Work flow graphs for Titan, Pathfinder, and the Virtual Microscope are illustrated in Figure 2.

The basic skeleton of a work flow graph is independent of specific instances of input data, output data and machine configuration. This skeleton depends on the characteristics of the application. The skeleton is embedded in the simulator. However, we need to parameterize work flow graphs to reflect the behavior of the application for a specific instance of input data, output data and machine configuration. For example, the number of send operations performed on a data-block in Titan depends on the spatial extent of the data-block and the partitioning of the output data structure across the processors. Parameterization of work flow graphs is done by the application emulators.

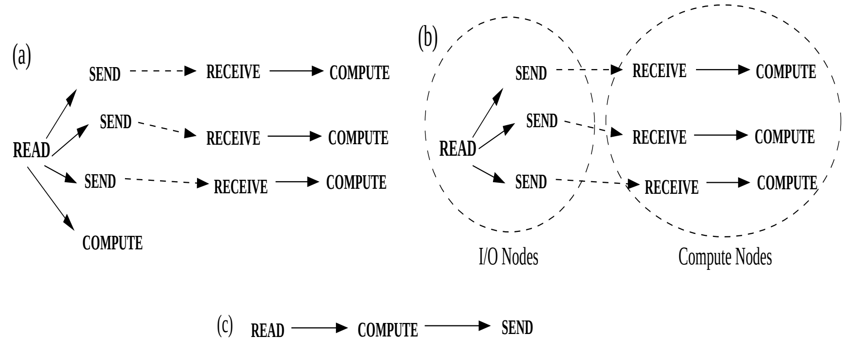


Figure 2: Work flow graphs for (a) Titan, (b) Pathfinder, (c) Virtual Microscope. The dashed circles for Pathfinder denote the operations performed on client and server nodes. Solid arrows denote dependencies within a processor, while dashed arrows denote inter-processor dependencies

4.3.2 Epoch-based Interaction between Emulator and Simulator

In epoch-based interaction, the emulator and the simulator interact in distinct phases, called *epochs*, instead of interacting for each event. In an epoch, the emulator passes the simulator a set of events for each simulated processor at the same time, along with the dependency information shown in Figure 2. The simulator then processes all the events, without invoking the emulator for each event. The simulator is responsible for determining the order in which these events are executed, while enforcing the dependencies. As a result, the emulator does not need to keep track of completed and outstanding events. When the simulator finishes processing all the events for a particular processor, it requests another set of events for that processor from the emulator. There are two main reasons for having the emulator pass the events in epochs rather than passing all the events at once. The first reason is to decrease the memory requirements of the simulator by limiting the number of events passed at the same time. The second is to give some control also to the emulator over the order in which events are executed. The operations represented by events in distinct epochs cannot be executed concurrently. This gives the emulator the ability to impose dependencies that cannot be represented as work flow graphs. For example, in the Virtual Microscope, processing of separate queries is currently not overlapped. The emulator generates events for these queries in distinct epochs, thus imposing this dependency.

In an epoch, the emulator passes the simulator the events for a set of data-blocks for each simulated processor. For the current suite of applications, the emulator encodes the following information for each data-block: *disk number*, *block size*, *list of consumer processors*, and *data-block computation time*. The *disk number* denotes the local disk where the block is stored, and implicitly indicates a read operation for that disk. The *block size* is used to estimate I/O and communication time. The *list of consumer processors* denotes the processors that will process this data-block (including the processor doing the I/O for the data-block if the block is processed locally). Thus, the list encodes the send operations to be issued for the data-block. It also implicitly encodes the corresponding receive and compute operations, since a send operation always has a matching receive and a received block is always processed (see Fig. 2).

Note that the total number of events generated and processed are the same for both the loosely-coupled and tightly-coupled simulation models. However, the order in which these events are processed may be different because of the different interactions between the simulator and application emulator in each model. To evaluate the efficiency and accuracy of these two approaches, we have developed two simulators, LC-SIM and TC-SIM, for loosely-coupled and tightly-coupled simulation, respectively. Both simulators employ the same hardware models, differing only in the way that they interact with the emulator.

5 Experimental Evaluation of Simulation Models

In this section, we present an experimental evaluation of our simulation models. We focus on simulation speed and the accuracy of the models as the machine configuration and datasets for the application scale. Our target platforms are distributed memory machines in which each node has local disks.

To evaluate the accuracy of the simulation models, we used two IBM SP machines with different configurations. The first machine is a 16-node IBM SP2 at the University of Maryland (UMD). Each node of this machine has a peak performance of 266 MFlops, 128 MB of main memory and six local disks, which are connected to the memory bus with two fast-wide SCSI buses (20 MB/s) – 3 disks on each bus. Nodes are interconnected through the High Performance Switch (HPS) with 40 MB/s peak bandwidth per node. In these experiments, the data sets were distributed across 15 nodes of the machine, on four disks per node. The second machine is a 128-node SP at the San Diego Supercomputing Center (SDSC). Each node of this machine has a peak performance of 640 MFlops, 256 Mbytes of main memory and one disk. All nodes are interconnected by a newer version of the HPS with 110 MB/s peak bandwidth per node. In the validation experiments described below, we have compared the execution times of the application emulators estimated by the simulators with the actual execution times of the application emulators running on these machines. Over all our experiments, the number of data-blocks for the Titan and Pathfinder applications was varied from 3.5K (4 days of satellite data) to 14115K, (2.6 Terabytes for 42 years of data). For the Virtual Microscope, the number of blocks was scaled from 5K (2 slides, 1 focal plane each) to 128000K blocks (5120 slides, 10 focal planes each) of size 384 terabytes. For Titan, we have used the "world query" that accesses all the blocks [6]. For the Virtual Microscope emulation, we generated a set of random queries. The number of simultaneous queries processed was scaled with the number of nodes in the machine.

Table 1 shows the validation results for TC-SIM and LC-SIM on the UMD SP. For Pathfinder, we fixed the number of compute nodes per I/O node to three in all measurements. The values in parentheses indicate the percent error in estimated execution times and are calculated as ratio of the difference between real execution time and estimated execution time to real execution time. As is seen from the table, the error of the predictions versus the actual execution times remains under 9% for TC-SIM and under 13% for LC-SIM for all the applications. As expected TC-SIM is more accurate than LC-SIM, because the interaction between the emulator and simulator more closely resembles the interaction between emulator and the real machine.

Figures 3 (a)-(d) show the validation results for LC-SIM on the SDSC SP. For the Titan application, we ran two different scenarios with the application emulators. For the first scenario, the input data size is scaled with the machine size (Fig. 3(a)). In this case, the input data size was varied from 14K data-blocks (16 days of satellite data) to 91K data-blocks (100 days). For the second scenario, the input data size is fixed at 14K data-blocks and the machine size is scaled (Fig. 3(b)). For the Pathfinder application, we varied the ratio of I/O nodes to compute nodes on 64 processors

Emulator	Data set	IBM SP2	TC-SIM	LC-SIM
Titan	9K blocks	113	105 (7%)	100 (12%)
	27K blocks	347	322 (7%)	306 (12%)
Pathfinder	9K blocks	166	153 (8%)	149 (10%)
	27K blocks	497	467 (6%)	452 (9%)
Virtual	5K blocks (200 queries)	127	122 (4%)	119 (6%)
Microscope	7.5K blocks (400 queries)	243	236 (3%)	234 (4%)

Table 1: Accuracy of the simulation models. All timings are in seconds. IBM SP2 figures represent the actual execution time of the emulators on 15 nodes of the UMD SP. The numbers in parentheses denote the percent error in estimating the actual runtime.

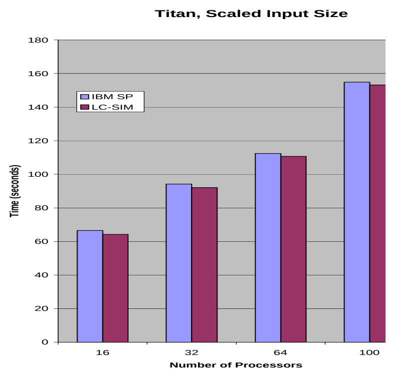
(Fig. 3(c)). The number of data-blocks was fixed at 3.5K, which was the largest number of data blocks that could be stored on the machine configuration with the smallest number of I/O nodes. For the Virtual Microscope application, we scaled the number of data-blocks from 10K to 64K, and scaled the number of queries from 160 to 1000 along with the machine size (Fig. 3(d)). As is seen from the figures, the estimated execution times are very close to the actual execution times for all application scenarios and for all machine sizes. The percent error remains below 4% for all cases. Our validation results show that we do not lose much from accuracy from using the loosely-coupled simulation model.

The execution times for the simulators for up to 128 processors are presented in Table 2. We ran our simulations on Digital Alpha 2100 4/275 workstations with 256 MB of memory. For LC-SIM, the application emulator was run on one workstation, while the simulator was run on a different workstation. The data exchange between the two programs was carried out using the Unix socket interface. As is seen in the table, TC-SIM executes for almost one hour to simulate even the smallest machine configuration. It runs for more than 32 hours for performance estimation of Titan on 128 processors. This shows that TC-SIM is only feasible for simulating a fairly small numbers of processors. LC-SIM, on the other hand, can simulate all machine configurations, even with very large datasets, in very little time (less than two minutes).

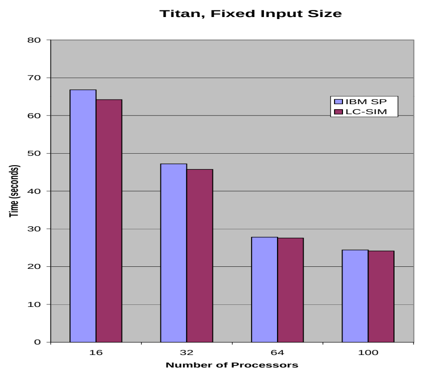
The execution times for LC-SIM when simulating very large scale machines are displayed in Table 3. We were able to do performance predictions for very large machines (8K processors, 32K disks) running applications with very large datasets (up to 384 terabytes) in less than 22 hours. These results show that performance prediction for large scale machines can be done in a reasonable amount of time on workstations using LC-SIM.

6 Related Work

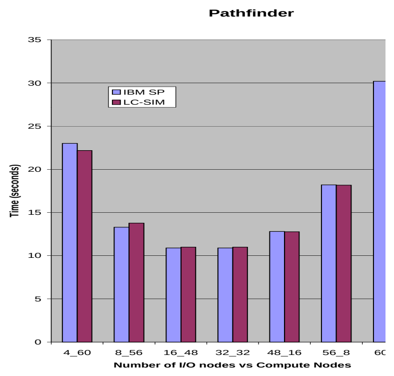
Performance prediction of applications on parallel machines is a widely studied area. Previous work in this area mainly focused on performance prediction of compute intensive scientific applications, but has taken several approaches. In [4, 7, 15], applications are modeled by a set of equations as a function of size of the input and number of processors. In [16, 17], applications are modeled as directed graphs (i.e. task graphs). The graph representation models the data and control flow in the application. The performance estimation is done by traversing the graph. Although these approaches are fast, so are feasible for large machine configurations and datasets, it is very difficult



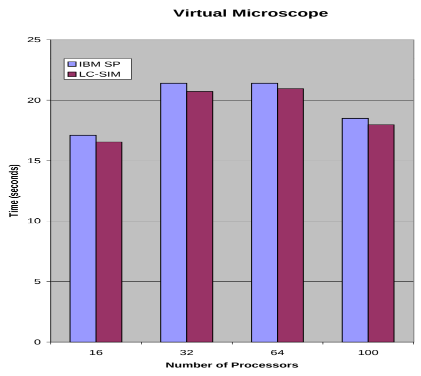
(a)



(b)



(c)



(d)

Figure 3: Accuracy of LC-SIM when simulating the application emulators running on the SDSC IBM SP machine. All results are in seconds. (a) Titan application, scaled input size. (b) Titan application, fixed input size. (c) Pathfinder, fixed input size, varying the ratio of IO nodes to compute nodes. The x-axis labels show the number of I/O nodes versus number of compute nodes. (d) Virtual microscope, scaled input data size and number of queries.

to model the dynamic and data dependent nature of applications (such as the ones we want to model) by equations or graphs. In addition, the graph modeling the application may grow very large for large scale machines. Traces obtained from application runs are used in [9, 11]. The main drawback of using traces is that a trace only represents the behavior of the application on a particular configuration of the machine, and cannot be used when the machine configuration is changed. An alternative approach to using traces or analytical models is to use simulators that run application programs. A variety of such simulators are available [3, 10, 13, 14]. In general, application source or binary codes must be augmented or the application is required to use the simulator API so that events can be passed to the simulator so that simulated time can progress. In order to increase the efficiency of simulation, most simulators use the *direct-execution technique*, in which the application code is executed on the host machine on which the simulator runs, and only the operations that cannot be run on the host machine, plus other events of interest, are captured by the simulator and simulated. Moreover, the simulators can employ less detailed architectural

Emulator	Data set	P	TC-SIM		LC-SIM	
			Estimated Application Time	Execution Time of Simulation	Estimated Application Time	Execution Time of Simulation
Titan	27K blocks	32	211	3426	182	6
	55K blocks	64	285	13154	217	14
	110K blocks	128	604	116224	420	28
Pathfinder	55K blocks	32	551	11595	496	22
	110K blocks	64	718	30446	579	57
	220K blocks	128	1020	97992	881	126
Virtual Microscope	500K blocks	32	135	7155	118	4
	1000K blocks	64	145	14097	126	8
	2000K blocks	128	158	37534	138	17

Table 2: Execution times of TC-SIM and LC-SIM. Estimated application times and simulator execution times are in seconds.

P	Titan		Pathfinder		Virtual Microscope	
	Estimated Application Time	Execution Time of Simulation	Estimated Application Time	Execution Time of Simulation	Estimated Application Time	Execution Time of Simulation
256	1147	172	1579	270	136	36
512	2276	520	2342	685	137	78
1024	4525	1454	4621	1762	144	189
2048	9031	4897	9177	5032	136	498
4096	18028	16388	18274	24562	142	1481
8192	36035	77641	36437	65137	148	4873

Table 3: Execution times for loosely coupled simulation. Both estimated and execution times are in seconds.

models for less accurate but fast simulation, or use parallel machines to run the simulators [3, 13].

Our work differs from previous work in several ways. In our work we specifically target large scale data-intensive applications on large scale machines. The application emulators presented in this paper lie in between pure analytical models and full applications. They provide a simpler, but parameterized, model of the application by abstracting away the details not related to a performance prediction study. Since an application emulator is a program, it preserves the dynamic nature of the application, and can be simulated using any simulator that can run the full application. The loosely-coupled simulation model reduces the number of interactions between the simulator and the application emulator by embedding the application processing structure into the simulator. As our experimental results show, our optimizations enable simulation of large scale machines on workstations.

7 Conclusions

In this paper, we have presented a performance prediction framework for data-intensive applications running on large scale machines. We have have implemented a performance prediction tool, which contains two components; *application emulators* and *architecture simulators*. We have developed application emulators that capture data access and computation patterns for three data-intensive applications. Emulators allow the critical components of applications to be changed and scaled easily, enabling performance prediction studies for large scale configurations. We have developed simulation models that are both sufficiently accurate and execute quickly. We presented a new technique, called *loosely coupled simulation*, that makes it possible to simulate large scale machines (up to several thousands of processors) inexpensively and accurately. Our preliminary results are very encouraging. We were able to model application datasets of up to 384 terabytes in size and run simulations that involve 8K processors on typical high performance workstations.

Acknowledgements

We would like to thank Jeff Hollingsworth and Hyeonsang Eom for their invaluable discussions about performance prediction on large scale machines.

References

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proc. of IOPADS'96*. ACM Press, May 1996.

[2] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):962–9, Dec. 1996.

[3] R. Bagrodia, S. Docy, and A. Kahn. Parallel simulation of parallel file systems and I/O programs. In *Proceedings of the 1997 ACM/IEEE SC97 Conference*. ACM Press, Nov. 1997.

[4] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy. PerPreT - a performance prediction tool for massively parallel systems. In *Proceedings of the Joint Conference on Performance Tools / MMB 1995*, pages 284–298. Springer-Verlag, Sept. 1995.

[5] C. F. Cerco and T. Cole. User's guide to the CE-QUAL-ICM three-dimensional eutrophication model, release version 1.0. Technical Report EL-95-15, US Army Corps of Engineers Water Experiment Station, Vicksburg, MS, 1995.

[6] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a high-performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering*, Apr. 1997.

[7] M. J. Clement and M. J. Quinn. Using analytical performance prediction for architectural scaling. Technical Report TR BYU-NCL-95-102, Networked Computing Lab, Brigham Young University, 1995.

[8] R. Ferreira, B. Moon, J. Humphries, A. Sussman, J. Saltz, R. Miller, and A. Demarzo. The Virtual Microscope. In *Proc. of the 1997 AMIA Annual Fall Symposium*, pages 449–453. American Medical Informatics Association, Oct. 1997.

[9] A. Gürsoy and L. V. Kalé. Simulating message-driven programs. In *Proceedings of Int. Conference on Parallel Processing*, volume III, pages 223–230, Aug. 1996.

[10] S. Herrod. Tango lite: A multiprocessor simulation environment. Technical report, Computer Systems Laboratory, Stanford University, 1993.

[11] C. L. Mendes. Performance prediction by trace transformation. In *Fifth Brazilian Symposium on Computer Architecture*, Sept. 1993.

[12] B. Moon and J. H. Saltz. Scalability analysis of declustering methods for multidimensional range queries. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):310–327, March/April 1998.

[13] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proc. of the 1993 ACM SIGMETRICS Conf. on Measuring and Modeling of Computer Systems*, 1993.

[14] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[15] J. M. Schopf. Structural prediction models for high-performance distributed applications. In *Cluster Computing Conference*, 1997.

[16] J. Simon and J.-M. Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of Euro-Par'96*, volume 1124 of *LNCS*, pages 675–688. Springer-Verlag, Aug. 1996.

[17] Y. Yan, X. Zhang, and Y. Song. An effective and practical performance prediction model for parallel computing on non-dedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38:63–80, Oct. 1996.